Open-Vocabulary Models for Source Code (Extended Abstract)

Rafael-Michael Karampatsis
University of Edinburgh
Edinburgh, United Kingdom

Hlib Babii
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy

Romain Robbes
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy

Charles Sutton
Google AI, University of Edinburgh
and The Alan Turing Institute
Mountain View, CA, United States

Andrea Janes
Free University of Bozen-Bolzano
Bozen-Bolzano, Italy

ABSTRACT

Statistical language modeling techniques have successfully been applied to large source code corpora, yielding a variety of new software development tools, such as tools for code suggestion, improving readability, and API migration. A major issue with these techniques is that code introduces new vocabulary at a far higher rate than natural language, as new identifier names proliferate. Both large vocabularies and out-of-vocabulary issues severely affect Neural Language Models (NLMs) of source code, degrading their performance and rendering them unable to scale.

In this paper, we address this issue by: 1) studying how various modelling choices impact the resulting vocabulary on a large-scale corpus of 13,362 projects; 2) presenting an *open vocabulary* source code NLM that can scale to such a corpus, 100 times larger than in previous work, and outperforms the state of the art. To our knowledge, this is the largest NLM for code that has been reported.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Naturalness of code, Neural Language Models, Byte-Pair Encoding

ACM Reference Format:
Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Open-Vocabulary Models for Source Code (Extended Abstract). In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), October 5–11, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3377812.3390806

1 INTRODUCTION

Previous work makes use of the "naturalness" of software [4] to build Language Models (LMs) that assist in a number of software engineering tasks. When training an LM, one of the decisions to be made is how to model vocabulary. The difference between natural language and code is that developers are allowed to create arbitrary complex identifiers. This leads to a large and sparse vocabulary, which makes it difficult to train an LM, in particular, a Neural LM (NLM). First, such vocabulary contains a lot of infrequent tokens. For each word in the vocabulary, an *embedding*, its continuous multi-dimensional representation, is learned. Infrequent words are rarely seen during the training, and therefore their embeddings are not learned well [2]. Second, many tokens in the test set are *out-of-vocabulary (OOV)*, i.e. they are not present in the training set, which makes it impossible to predict them. Third, the dimensions of the first layer of the neural network, formed by the embedding vectors (embedding layer), and the output layer, which returns the probabilities over the whole vocabulary, become extremely large with the large vocabulary. As a result, the model is computationally hard to train. Hellendoorn and Devanbu showed that Neural Language models (which are a state-of-the-art approach in NLP) are not able to scale beyond a few hundreds of projects [3].

2 VOCABULARY MODELING

To select a vocabulary modeling approach that makes it possible to train a high-performing NLM that can scale, we evaluate different modeling choices. Since having a small vocabulary, frequent tokens, and keeping OOV low is so important, we primarily consider these characteristics of the vocabulary during the evaluation.

For evaluation we use 13,362 projects. Our baseline is the vocabulary of unsplit tokens, which is extremely large (11.6M tokens) and greatly exceeds 75,000 tokens, which is considered in the previous work [3] to be a limit beyond which it is not possible to train an NLM. There is a significant number of infrequent tokens (83% of tokens in the vocabulary that occur only 1 to 10 times). The OOV rate is also high (42%). To calculate the OOV rate, we build vocabulary on a held-out set of 38 projects and count the percentage of tokens in it which were not encountered in our large set.

We start exploring modeling choices by filtering different categories of tokens one by one: tokens containing non-ASCII characters, whitespace, comments, and string literals. Then we split compound identifiers by conventions and encode case information in special tokens *<Upper>*, *<UPPER>*. After that, we split numbers and apply stemming. Even all the techniques above, combined, are not able to bring the vocabulary to a manageable size (it is still in the hundreds of thousands) and handle frequency and OOV issues.

We considered representing tokens as sequences of characters, which makes the vocabulary, OOV, and frequency issues vanish.

ICSE '20 Companion, October 5–11, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3390806

However, character-level models need to learn much longer sequences and were shown to perform worse than their subtoken-level counterparts [6].

The final choice we evaluate is Byte Pair Encoding (BPE) [1] [7]. BPE first splits tokens into sequences of characters and then on each iteration merges the sequences that occur together most often. As a result, infrequent tokens are split into sequences of frequent subtokens (e.g. `http • client • lib`), frequent ones remain as they are (e.g. `toString`). This fits our needs since the vocabulary size can be controlled by specifying the number of merges which is given as input to the BPE algorithm. Moreover, there are no more OOV; all the subtokens are frequent.

3 NEURAL LANGUAGE MODEL FOR CODE

Since we use BPE to split tokens, our NLM operates on the level of subwords rather than words, i.e. it predicts a subword at each step. To make the model output full words, we use a variation of the beam search algorithm. As a model's architecture, we choose a single-layered GRU (a variant of an RNN capable of handling long dependencies). We train models with 512 and 2,048 features.

To make use of the locality of source code, we use the cache mechanism: for each token encountered at test time we store its 5 proceeding tokens. To make a prediction, the sequence of the last 5 tokens is looked up in the cache and probabilities are assigned to the token retrieved from cache. These probabilities contribute to the probabilities generated by the beam search.

Finally, we dynamically adapt the global model to the current project by training the model on it for only one epoch. This makes this adaptation step fast, and also prevents the model from "forgetting" what it has learned already.

4 EVALUATION AND RESULTS

We compare our BPE NLMs with two closed-vocabulary NLMs with the same architecture (one trained on full tokens; the other trained on tokens split by conventions) and n-gram models [3]. We train each of our NLMs both on a set of 107 projects (small training set) and 13,362 (large training set).

We evaluate the models in static, dynamic, and maintenance scenarios from the previous work [3] on Java, C, and Python corpora for the tasks of code completion and bug prediction. In this paper, we present only the results for code completion in a dynamic scenario on the Java corpus and report mean reciprocal rank (MRR) as the only metric. For all results please refer to our full paper [5]. In the dynamic scenario, the model can update its parameters after it makes a prediction. After each project, the model restores its weight to the values it had before testing. MRR is the average over the multiplicative inverse values of the ranks of the correct predictions. In other words, it is the average of top-k accuracies across various k, e.g., a correct prediction at rank 1 yields an MRR of 1; at rank 2, 0.5; at rank 10, 0.1.

RQ1. How does the performance of subword unit NLMs compare to state-of-the-art LMs for code? Already on the small training set, our BPE NLM with 512 features reaches an MRR of 77.02% and outperforms the state-of-the-art nested cache n-gram LM [3] (74.55%) as well as the closed-vocabulary models (64.05% - 71.01%). Adding more features makes the MRR increase by 0.28%. The use of cache improves the performance further by 0.99%.

RQ2. Can subword unit NLMs scale to large code corpora? Does the additional training data improve performance? The performance of the n-gram models does not improve significantly with the increase of training data [3]. NLMs, on the contrary, do improve significantly (MRR rises from 77.02% to 79.94%). The NLM with 2,048 features can leverage the large training set even better due to its larger capacity (increase in MRR from 77.30% to 82.41%). The cache still increases the performance (by 0.35% to 0.86%). All in all, our best NLM with cache reaches the MRR of 78.29% and outperforms the n-gram nested cache model by 8.72%. Furthermore, when scaling n-gram models to the large training set, not only the performance is a problem but also the resource usage. They require 50 to 60 GB of RAM on the large training set, which makes them unusable in practice. On the other hand, for BPE NLMs, the required RAM when doing code completion ranges from 250 to 400 MB.

Other results. Our observations also hold for C and Python, and for different vocabulary sizes (2k, 5k, and 10k). Besides, we discovered that our dynamic adaptation is extremely effective, especially on the small training set (up to 13% of MRR increase) and achieves the state of the art. Notably, our NLMs outperform other models not only at the code completion task but also at predicting bugs, particularly when trained on a large corpus.

5 CONCLUSIONS

We evaluated different vocabulary modeling choices and concluded that the only viable option to make NLM applicable in practice is to use BPE. We also presented a subtoken BPE open vocabulary NLM which unlike closed vocab models is able to scale for large corpora. Our NLM uses dynamic adaptation, beam search and caching to successfully predict next tokens. We evaluated the model in different scenarios and showed that it outperforms the state of the art. We hope that our model, which to our knowledge is the largest NLM for code ever trained, will open possibilities for transfer learning and building new tools to aid software engineering.

Acknowledgements. This work was supported by the EPSRC Centre for Doctoral Training in Data Science, the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1), the University of Edinburgh, the Free University of Bozen-Bolzano (IDEALS and ADVERB projects), and the Vienna Scientific Cluster (VSC).

REFERENCES

[1] Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal* 12, 2 (1994), 23–38.
[2] Chengyue Gong, Di He, Xu Tan, Tao Qin, Liwei Wang, and Tie-Yan Liu. 2018. Frage: Frequency-agnostic word representation. In *Advances in neural information processing systems*. 1334–1345.
[3] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proc. of ESEC/FSE 2017*. 763–773.
[4] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of ICSE 2012*. IEEE, 837–847.
[5] Rafael Karampatsis, Hlib Babii, Andrea Janes, Charles Sutton, and Romain Robbes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of ICSE 2020*. in press.
[6] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. 2012. Subword language modeling with neural networks. *preprint (http://www.fit.vutbr.cz/ imikolov/rnnlm/char.pdf)* (2012).
[7] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).